# JPF Application Goals and Types

By now, you already know that you run JPF on compiled Java programs, just like a normal VM. But what are the goals to apply JPF, and - based on these goals - what are the different JPF application types?

## Why to Use JPF?

Before we classify the different types of JPF applications, it is worth to spend a few thoughts on why we actually want to apply it. A word of caution: if you have a strictly sequential program with only a few well defined input values, you are probably better off writing a few tests - using JPF won't tell you much. There are two major reasons to run JPF:

### Explore Execution Alternatives

After all, JPF started as a software model checker, so its original domain is to explore execution choices, of which we have four different types:

- scheduling sequences - concurrent applications are still the major domain for JPF application because (a) defects like deadlocks and data races are subtle and usually spurious, and (b) the scheduler can usually not be controlled from a testing environment, i.e. this is hard to impossible to test. JPF on the other hand not

only "owns" the scheduler (it's a Java runtime) and explores all interesting scheduling combinations, it even let's you define your own scheduling policies
- input data variation - JPF allows you to explore input value sets that can be defined by heuristics (e.g. a value below, at, and above a certain threshold). This is esp. useful to write test drivers
- environment events - program types like Swing or web applications usually react to external events like user input. These can be simulated by JPF as choice sets
- control flow choices - JPF can not only check how your program reacts to a concrete input, it can also turn around and systematically explore the program control structure (branch instructions), e.g. to derive interesting test data values

### Execution Inspection

Even if your program does not have a lot of execution alternatives, you can make use of JPF's inspection capabilities. Being an extensible JVM, it is relatively easy to implement coverage analyzers, or non-invasive tests for conditions that would otherwise go unnoticed (like overflow in numerical instructions).

## JPF Application Types

There are three basic JPF application types, and each of them has different strengths and weaknesses: JPF- aware, unaware, and "enabled" programs.

Figure: JPF application types

### JPF unaware programs

This is the usual case - you run JPF on an application that is not aware of verification in general, or JPF in particular. It just runs on any VM that is Java compatible. The typical reason to check such an application with JPF is to look for violations of so called non-functional properties that are hard to test for, like deadlocks or race conditions. JPF is especially good at finding and explaining concurrency related defects, but you have to know the costs: JPF is much slower than a production VM (for a reason - it does a lot more), and it might not support all the Java libraries that are used in the application (which usually is easy to add)

### JPF dependent programs

On the other side of the spectrum we have applications that are models - their only purpose in life is to be verified by JPF (e.g. to check a certain algorithm), so Java just happens to be the implementation language because that's what JPF understands. Typically, these applications are based on a domain specific framework (like the [[ext:statechart|UML statechart extension]]) that has been written so that JPF can handle the model, and knows what to look for. As a consequence, the model applications themselves are usually small, scale well, and do not require additional property specification. The downside is that it is quite expensive to develop the underlying domain frameworks.

## JPF enabled programs

The third category probably has the best return on investment - programs that can run on any VM, but contain Java annotations that represent properties which cannot easily be expressed with standard Java language.

For example, assume you have a class which instances are not thread safe, and hence are not supposed to be used as shared objects. You can just run JPF and see if it finds a defect (like a data race) which is caused by illegally using such an object in a multi-threaded context. But even if JPF out-of-the-box can handle the size of the state space and finds the defect, you probably still have to spend significant effort to analyze a long trace to find the original cause (which might not even be visible in the program). It is not only more easy on the tool (means faster), but also better for understanding if you simply mark the non-threadsafe class with a *@NonShared* annotation. Now JPF only has to execute up to the point where the offending object reference escapes the creating thread, and report an error that immediately shows you where to fix it.

There is an abundance of potential property-related annotations, which can even be useful for several tools (e.g. *@NonNull* for a static analyzer and JPF), including support for more sophisticated, embedded languages to express pre- and post-conditions.